Intermediate iRules: Handling Strings

Jason Rahm, 2016-07-06

To say we're getting to the heart of the matter, dealing with string commands and parsing, re-arranging and modification, would almost be saying it too lightly…understating. String manipulation is a massive part of iRules, and is in fact a solid part of why we are using Tcl as our language of choice, along with many others that I've covered elsewhere in detail. String manipulation is useful in many ways, in many places. Whether it's re-writing a URI for an inbound HTTP request or parsing part of the TCP payload and twiddling the bits to read a bit differently, or perhaps just determining the first n characters of a string to be used for some purpose or another…all of it revolves around strings.

Fortunately for us, Tcl handles strings extremely well. In fact, everything is a string in Tcl's eyes, and as such there are many powerful tools with which you can twist strings to your desires with relative ease, and great effect. To cover all of the options would be a huge process, but we'll go over the basics here, the things seen most commonly within iRules, and you can research the more obscure wizardry at will. The publicly available documentation is good for most of the commands in question. So, in this article we will cover what a string is and why you should care, as well as a large subset of the string commands you're most likely to use.

What is a string and why do I care?

A string is a particular data type, and is generally understood to be a sequence of characters, either as a literal constant, or represented in variable form. This means that basically anything can be a string. A name, an IP address, a URL…all of them are strings, especially in Tcl. Generally speaking, unless things are specifically typed as an integer or some other data type, it is a safe bet to assume they are a string. That being said, since Tcl is not a statically typed language and thereby does not allow you to specify data types explicitly, it treats everything as a string save for a few specific conditions. This is a good thing for iRules, as it means that there isn't a lot of messing about with data types, and that you can generally manipulate things in string format without much hassle. That means less programming fuss, and more getting the effect you want.

What are the most commonly used string commands?

First off, the most common and widely used command in Tcl for dealing with strings is, quite simply, "string". Mind you, in and of itself this command has little use. There are many, many permutations of this command from changing the case of a string to referencing only a portion of it, to re-ordering it and more. With this single command, and the many sub commands, you can perform the lion's share of your string work within iRules. So the question is really which "string" sub commands are most commonly used?

This one is a bit of an intense, broad sweeping question. There are so many things that you can do with a string in Tcl that listing them all here would be an indigestible amount of information, and wouldn't make sense to portray. As such, I'll do my best to list a few string commands that seem to often crop up in iRules, and discuss what each does. For a full reference on the string command and other Tcl base commands, you can find the official Tcl documentation online here (http://www.tcl.tk/man/tcl8.4/TclCmd/contents.htm)

string tolower

`string tolower string ?first? ?last?`

Without question the most common and widely used string command within iRules is also one of the simplest. The `tolower` command does pretty much what it sounds like. It converts the entirety of a string's contents to lowercase. Meaning, if you had a variable named `$uri` and the contents were, "/Admin/WebAccess", you could run the `string tolower` command when performing comparisons to get a different result.

For instance:

```
set uri "/Admin/WebAccess"
log local0. "Uri : $uri"
```

```
log local0. "Lower Uri: [string tolower $uri]"
```

Would result in "Uri: /Admin/WebAccess" for the first log message, and "Lower Uri: /admin/webaccess" for the second. This is thanks to the `string tolower` command. Why is this so useful in iRules? Because any time you're performing a string based comparison, it is important to be sure you're comparing things in the same case. Think about comparing a host name, a URI, etc. and suddenly you may see why there's so much value in this simple command. This becomes increasingly important with things like data groups, where you are comparing a single value against a broad range of key values. Being able to assure they are all in the proper case, and then force the incoming comparison value to that case is extremely useful.

Keep in mind that this, like most of the other string commands, does not actually modify the string itself. If you took our above example where we provided the lowercase URI and referenced $uri again, it would still maintain the original case, unaltered. For example, ensuring you directed users attempting to access the admin portion of an application while ensuring they aren't worried about proper casing gets simpler with the `tolower` command:

```
when HTTP_REQUEST {
 if {([HTTP::uri] starts_with "/admin") || ([HTTP::uri] starts_with "/Admin")} {
  pool auth_pool
 }
}
```

Becomes:

```
when HTTP_REQUEST {
 if {[string tolower [HTTP::uri]] starts_with "/admin"} {
  pool auth_pool
 }
}
```

string length

`string length string`

Much as you'd expect given the name, the string length command returns the length of the string in question. This can be used for many different things, but probably the most common use-case observed so far in iRules has been to ascertain whether or not a given command returned a proper result. For instance:

```
when HTTP_REQUEST {
 set cookie_val [HTTP::cookie "x-my-cookie"]
 if {[string length $cookie_val > 1} {
  log local0. "cookie was passed properly"
  pool http_pool
 }
}
```

Of course there are many ways to perform a similar check, and some are even more efficient if all you're trying to do is identify whether or not a command returned null or not, but if you want to check to see if a specific answer was set of at least n characters, or for a few other very handy purposes I've seen, the `string length` command can be handy.

string range

`string range string first last`

The `string range` command allows you to reference a particular portion of a given string and retrieve only that specific range of characters. This could be characters 1-10, the first character to the 3rd, or perhaps the 15th to the end of the string. There are many different ways to reference string segments and divide things up using this command, but the result is the same. It returns the value of the portion of the string you define.

This has proved useful time and time again in iRules for things like retrieving portions of a URI, ensuring that a hostname starts with a particular prefix, or dozens of other such seemingly simple requirements. Without the `string range` command those benign tasks would be a major headache. Note that the first character in the string starts with an ID of 0, not 1.

For instance, if you're looking at a URI that is "/myApp?user=bob" where bob is a variable username, and you're looking to return only the username you have a few options, but `string range` makes that quite simple:

```
when HTTP_REQUEST {
 set user [string range [HTTP::uri] 12 end]
 log local0. "User: $user"
}
```

This next example shows the removal of a non-standard port from the value returned by `HTTP::host`. Notice the use of end-5, which will use the range from character in the zero index through the character five short of the end of the string.

```
when HTTP_REQUEST {
  if { [HTTP::host] ends_with "8010" } {
    set http_host [string range [HTTP::host] 0 end-5]
    HTTP::redirect "https://$http_host[HTTP::uri]"
  }
}
```

string map

`string map mapping string`

Where `string range` allows you to select a given part of a string and return it, `string map` allows you to actually modify sub strings in-line. Also, instead of acting on a count or range of characters, `string map` works with an actual string of characters. Whereas with `string range` you may want to look up a particular part of a URI, such as the first 10 characters, and see if they match a string, or route based on them or…something; with `string map` you are able to make changes in real-time, changing one string of characters to another.

For instance with `string range` you may have a logic statement like "Do the first 10 characters of the URI match x". You'd supply the string to fetch the range from and the number of characters you want, by giving a beginning and end character. With `string map` you'd be saying something like "look for any string that looks like x, and change it to y in the given string" by providing the string to work against as well as a source and destination string, meaning "Change all cases of http to https".

```
when HTTP_RESPONSE {
 set new_uri [string map {http https} [HTTP::header "Location"]]
 HTTP::header replace Location $new_uri
}
```

Of note, the string is only iterated over once, so earlier key replacements will have no affect for later key matches. For example:

```
% string map {abc 1 ab 2 a 3 1 0} 1abcaababcababababc
01321221
```

What?? That's one of those not so intuitive examples in the TCL documentation. Actually, though, I like this one. Let's break it down. There are four key/value pairs here:

1. abc, if found, will be replaced by a 1 ab, if found, will be replaced by a 2 a, if found, will be replaced by a 3 1, if found, will be replaced by a 0

String Map Multiple Key/Value Example

Mapping	Original String	Resulting String

1st (abc->1)	1abcaababcababcabc	11aab1abab1
2nd (ab->2)	11aab1abab1	11a21221
3rd (a->3)	11a21221	113321221
4th (1->0)	11321221	01321221

Note that with the fourth map, the returned string is 01321221, not 00320220. Why is that? Well, the string is only iterated over once, so earlier key replacements will have no affect for later key matches.

string first

```
string first string1 string2 ?startIndex?
```

The `string first` command allows you to identify the first occurrence of a given sub string of characters within a string. This can be extremely useful for combining with the `string range` command. For instance, if I want to find the first occurrence of "/admin" in a URI and collect the URI from that point to the end, it would be quite difficult without the `string first` command. What if I don't know what the exact URI will be? What if there is a variable portion of the URI that comes before "/admin" that I don't want to collect, but have to somehow account for even though it is variable in length? I can't just set a `static range` and use the `string range` command alone, so I have to get creative and combine commands.

By making use of the `string first` command, if I have a URI that looks something like "/users/apps/bob/bobsapp?user=admin" where the username is variable length and I can't be certain of the length of the URI because of it, but I wanted to retrieve the user argument being passed in, I could do something like:

```
set user [string range [HTTP::uri] [expr {[string first "user=" [HTTP::uri]] + 5}] end]
```

What the above is doing is finding the first occurance of "user=" in the URI and returning the index of the first character. Then adding 5 to that, since that is the length of the string "user=", and we want to reference what comes after "user=", not include it, then take the range of the string from that point to the end, and return that as the value of the username being passed in. It looks a bit complex, but if you break it down command by command, you're really just stringing together several smaller commands to get the functionality you want. And now you can start to see why string commands are so important and powerful in iRules.

string last

```
string last needleString haystackString ?lastIndex?
```

Similar to `string first`, only returns the index of the first character in the last such match within haystackString. If there is no match, then return -1. If lastIndex is specified, then only the characters in haystackString at or before the specified lastIndex will be considered by the search. In this example, (combined with `string range`, the conditional returns true if the string beginning with the last "/" in the URI, and ending with the last character of the URI, contains a "."

```
when HTTP_REQUEST {
  if {[matchclass [HTTP::uri] ends_with $::unmc_extends] or
  [matchclass [HTTP::method] equals $::unmc_methods] or
  [matchclass [HTTP::uri] contains $::unmc_sql] or
  [matchclass [IP::client_addr] equals $::unmc_restrict_ips]}{
    discard
  } elseif {
  ([HTTP::uri] ends_with "/") or
([string range [string last / [HTTP::uri]] end] contains ".") or
  ([HTTP::uri] contains "unmcintranet")}{
    pool unmc-intranet-proxy
  } elseif {[HTTP::uri] contains "google"}{
    HTTP::redirect "http://[HTTP::host][HTTP::uri]&restrict=unmcintranet"
  } else {
    HTTP::redirect "http://[HTTP::host][HTTP::uri]/"
  }
```

```
}
```

string trim

`string trim string ?chars?`

Again, a command true to its name, the `string trim` command allows you to manipulate the beginning and end of a given string to trim off unwanted characters. Perhaps you have a string that you need to ensure doesn't begin or end with white space, or you're looking at URI comparisons and need to be sure that you don't have a trailing slash in some cases and not others. Regardless, the `string trim` command makes that easy. All you'd do is specify which characters you want to ensure are removed, either whitespace or slashes in the examples just mentioned, and you'd be set to ensure standardized comparisons. For instance, if you want to ensure that someone is making a request to the admin section of your domain you can be sure that they are going to have a slash at the beginning of the URI, but they may or may not include a trailing slash. You could use the `starts_with` comparison operator to ignore this, but that would also ignore anything else they may include after the specific string. If you want an exact match for either "/admin" or "/admin/" you could use an `or`, or could trim the URI, like so:

```
when HTTP_REQUEST {
 if{[string trim [HTTP::uri] "/"] eq "admin"} {
  pool admin_pool
 }
}
```

Note that by using the trim command it removed both the preceding and trailing slashes. There are `trimright` and `trimleft` versions as well, for only trimming one end of a string, if that's necessary.

Note: The source material for this article was pulled from articles authored by Joe Pruitt, Colin Walker, and Jason Rahm. They are now archived but can still be referenced if necessary.

F5 Networks, Inc. | 401 Elliot Avenue West, Seattle, WA 98119 | 888-882-4447 | f5.com

F5 Networks, Inc.
Corporate Headquarters
info@f5.com

F5 Networks
Asia-Pacific
apacinfo@f5.com

F5 Networks Ltd.
Europe/Middle-East/Africa
emeainfo@f5.com

F5 Networks
Japan K.K.
f5j-info@f5.com